**ЗАДАНИЕ**

Разберитесь как устроена функция syscall(). Напишите код, который напечатает hello world без использования функции syscall().

.data

str:

.ascii "Hello world\n"

len = . - str

.text

.global \_start

\_start:

movq $1, %rax // 1 - номер системного вызова write

movq $1, %rdi // 1 - файловый дескриптор stdout

movq $str, %rsi

movq $len, %rdx

syscall // write

movq $60, %rax // 60 - номер системного вызова sys\_exit

xorq %rdi, %rdi // Обнуляем rdi (код возврата 0)

syscall // exit

*.data*

этот сегмент объявляет раздел данных программы. в этом разделе обычно хранятся строки, числа и другие статические данные.

*str:*

*.ascii "Hello world\n"*

здесь создаётся строка с текстом "Hello world\n" и присваивается метке str.

**Метка** в ассемблере — это символ, который указывает на определённое место в памяти программы. Метка помогает компилятору и программисту указать, где в памяти будет находиться нужный участок данных. В данном случае, метка str говорит о том, где начинается строка "Hello world\n". Это важно, потому что в дальнейшем мы будем использовать её адрес в других частях программы.

*len = . - str*

вычисляется длина строки str. спецсимвол . обозначает текущую позицию в памяти (тот адрес, где сейчас находится процесс компиляции или сборки программы), а str — метку начала строки. вычитая их, мы получаем длину строки (в байтах).

*.text*

сегмент .text содержит инструкции программы. это основной код, который будет выполняться.

*.global \_start*

директива .global делает метку \_start доступной для линковщика, указывая, что с неё начинается выполнение программы.

*\_start:*

метка \_start — это точка входа программы, с которой начнётся её выполнение.

*movq $*1*, %rax*

команда movq перемещает значение 1 в регистр %rax. в системных вызовах Linux (в 64-битной архитектуре) значение в %rax определяет номер системного вызова. номер 1 соответствует системному вызову write, который используется для вывода данных на экран.

*movq $1, %rdi*

значение 1 записывается в регистр %rdi. в системном вызове write, регистр %rdi указывает на файловый дескриптор, куда будет произведён вывод. значение 1 означает стандартный вывод (stdout).

*movq $str, %rsi*

здесь в регистр %rsi загружается адрес строки, которую нужно вывести. это адрес переменной str, который указывает на строку "Hello world\n".

*movq $len, %rdx*

регистр %rdx получает значение длины строки, которая была вычислена ранее (метка len), что позволяет системному вызову понять, сколько байт нужно вывести.

syscall

вызывает системный вызов. в данном случае это системный вызов write, который, используя значения в регистрах %rdi, %rsi и %rdx, выводит строку на экран.

*movq $*60*, %rax*

после завершения вывода, мы готовим системный вызов для завершения программы. в %rax загружается значение 60, что соответствует системному вызову exit.

*xorq %rdi, %rdi*

команда xorq %rdi, %rdi обнуляет регистр %rdi (выполняет операцию XOR между регистром %rdi и самим собой), который в системном вызове exit указывает на код завершения программы. если код равен 0, программа завершится успешно.

syscall

инициирует завершение программы. после этого процесс завершится с кодом возврата 0.

**Запуск**:

as hello\_without\_syscall.s -o hello\_without\_syscall.o

opp11@comrade:~/OS/lab2$ ld hello\_without\_syscall.o -o hello\_without\_syscall

opp11@comrade:~/OS/lab2$ ./hello\_without\_syscall

Hello world

strace ./hello\_without\_syscall

execve("./hello\_without\_syscall", ["./hello\_without\_syscall"], 0x7ffcf07538d0 /\* 71 vars \*/) = 0

write(1, "Hello world\n", 12Hello world

execve — системный вызов, который загружает и исполняет новый процесс.

write(1, "Hello world\n", 12) — это системный вызов sys\_write, который записывает данные в файловый дескриптор.

exit(0) — системный вызов sys\_exit, который завершает программу.

**ТЕОРИЯ**

В x86-64 используется syscall вместо int $0x80. Различия:

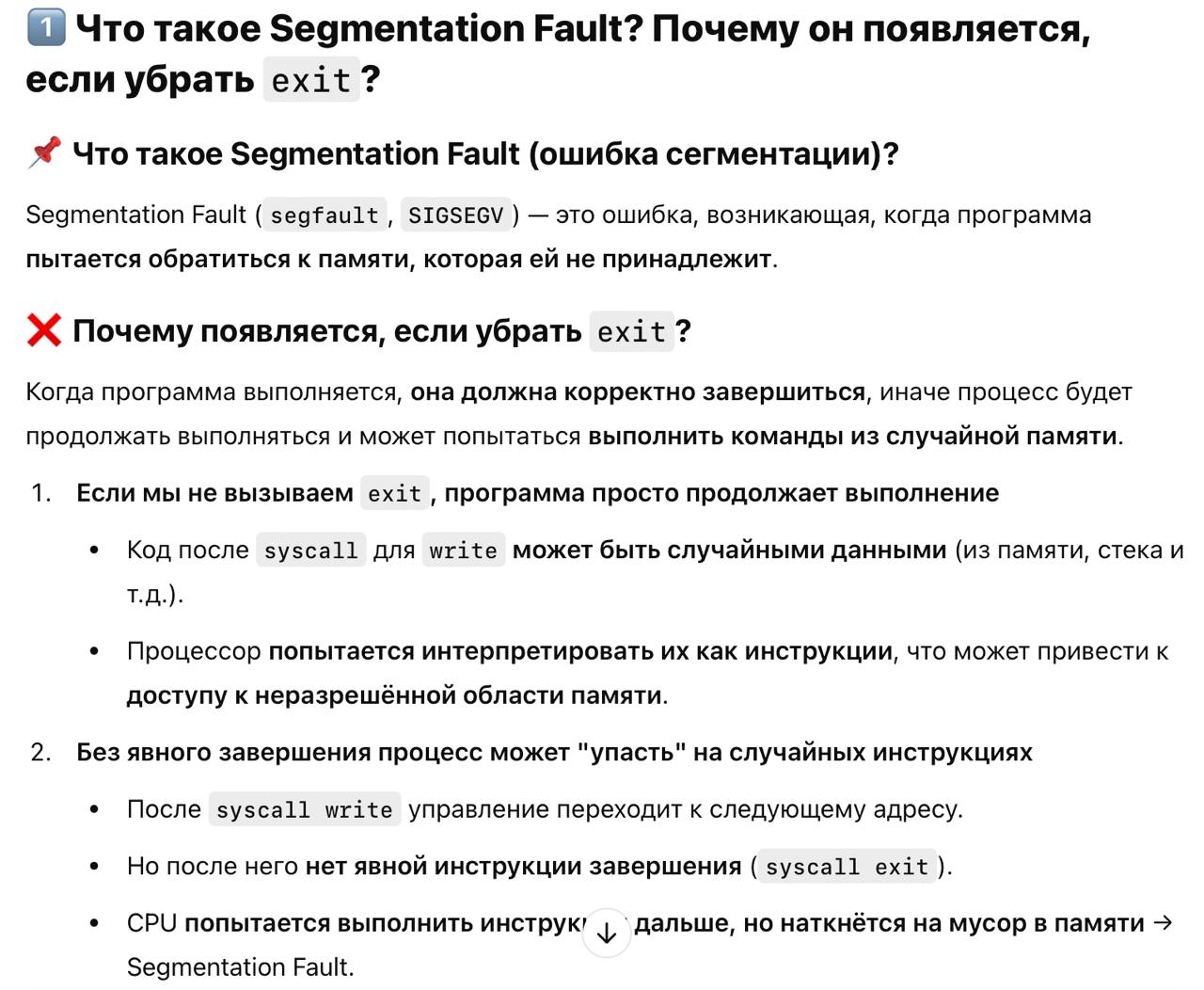
1. syscall выполняется быстрее, так как это специальная инструкция, а int $0x80 — программное прерывание, требующее дополнительной обработки.

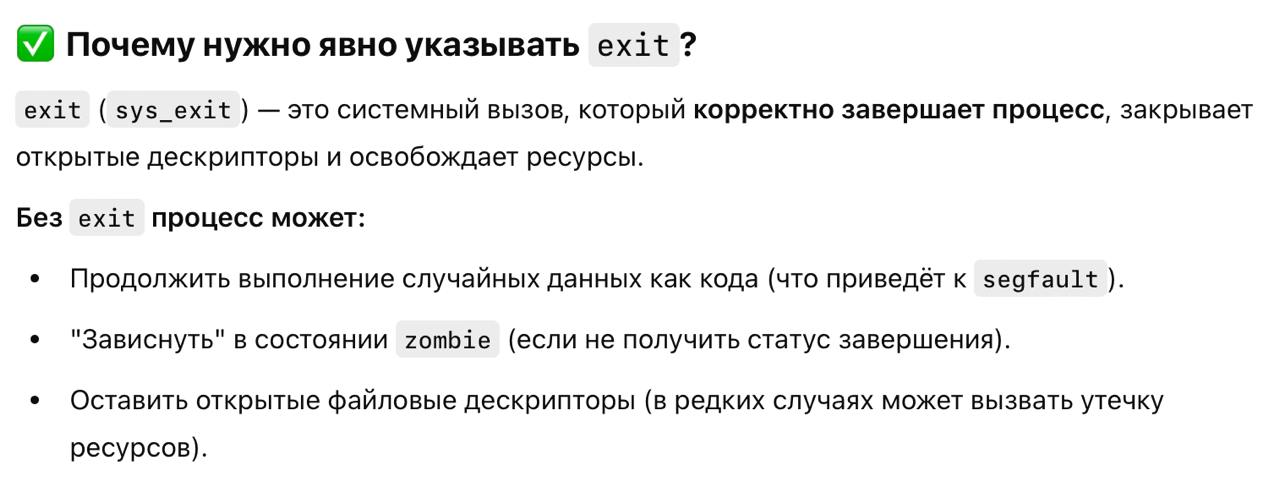
2. int $0x80 работает в x86-64, но требует сохранения и восстановления некоторых регистров. syscall использует специальную системную область памяти, что делает его эффективнее.

3. В int $0x80 аргументы передаются через ebx, ecx, edx... В syscall используются rdi, rsi, rdx..., что удобнее в 64-битных системах.

4. Современные ОС (Linux x86-64) используют syscall

5. Хотя int $0x80 все ещё работает, он устарел, а в некоторых новых ядрах Linux может быть отключён.





Сравнение системных вызовов

